26 | 多个活动要安排（下）：如何实现进程的等待与唤醒机制？

2021-07-07 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨
时长 08:30 大小 7.80M

你好，我是 LMOS。

上节课，我带你一起设计了我们 Cosmos 的进程调度器，但有了进程调度器还不够，因为调度器它始终只是让一个进程让出 CPU，切换到它选择的下一个进程上去运行。

结合前面我们对进程生命周期的讲解，估计你已经反应过来了。没错，多进程调度方面，我们还要实现进程的等待与唤醒机制，今天我们就来搞定它。

这节课的配套代码，你可以从这里下载。

进程的等待与唤醒

我们已经知道，进程得不到所需的某个资源时就会进入等待状态，直到这种资源可用时，才会被唤醒。那么进程的等待与唤醒机制到底应该这样设计呢，请听我慢慢为你梳理。

进程等待结构

很显然，在实现进程的等待与唤醒的机制之前，我们需要设计一种数据结构，用于挂载等待的进程，在唤醒的时候才可以找到那些等待的进程 ，这段代码如下所示。

复制代码

```
typedef struct s_KWLST
{
    spinlock_t wl_lock;  //自旋锁
    uint_t   wl_tdnr;    //等待进程的个数
    list_h_t wl_list;    //挂载等待进程的链表头
}kwlst_t;
```

其实，这个结构在前面讲信号量的时候，我们已经见过了。这是因为它经常被包含在信号量等上层数据结构中，而信号量结构，通常用于保护访问受限的共享资源。这个结构非常简单，我们不用多说。

进程等待

现在我们来实现让进程进入等待状态的机制，它也是一个函数。这个函数会设置进程状态为等待状态，让进程从调度系统数据结构中脱离，最后让进程加入到 kwlst_t 等待结构中，代码如下所示。

复制代码

```
void krlsched_wait(kwlst_t *wlst)
{
    cpuflg_t cufg, tcufg;
    uint_t cpuid = hal_retn_cpuid();
    schdata_t *schdap = &osschedcls.scls_schda[cpuid];
    //获取当前正在运行的进程
    thread_t *tdp = krlsched_retn_currthread();
    uint_t pity = tdp->td_priority;
    krlspinlock_cli(&schdap->sda_lock, &cufg);
    krlspinlock_cli(&tdp->td_lock, &tcufg);
    tdp->td_stus = TDSTUS_WAIT;//设置进程状态为等待状态
    list_del(&tdp->td_list);//脱链
    krlspinunlock_sti(&tdp->td_lock, &tcufg);
    if (schdap->sda_thdlst[pity].tdl_curruntd == tdp)
```

```
    {
        schdap->sda_thdlst[pity].tdl_curruntd = NULL;
    }
    schdap->sda_thdlst[pity].tdl_nr--;
    krlspinunlock_sti(&schdap->sda_lock, &cufg);
    krlwlst_add_thread(wlst, tdp);//将进程加入等待结构中
    return;
}
```

上述代码也不难，你结合注释就能理解。有一点需要注意，这个函数使进程进入等待状态，而这个**进程是当前正在运行的进程，而当前正在运行的进程正是调用这个函数的进程**，所以一个进程想要进入等待状态，只要调用这个函数就好了。

进程唤醒

进程的唤醒则是进程等待的反向操作行为，即从等待数据结构中获取进程，然后设置进程的状态为运行状态，最后将这个进程加入到进程调度系统数据结构中。这个函数的代码如下所示。

复制代码

```
void krlsched_up(kwlst_t *wlst)
{
    cpuflg_t cufg, tcufg;
    uint_t cpuid = hal_retn_cpuid();
    schdata_t *schdap = &osschedcls.scls_schda[cpuid];
    thread_t *tdp;
    uint_t pity;
    //取出等待数据结构第一个进程并从等待数据结构中删除
    tdp = krlwlst_del_thread(wlst);
    pity = tdp->td_priority;//获取进程的优先级
    krlspinlock_cli(&schdap->sda_lock, &cufg);
    krlspinlock_cli(&tdp->td_lock, &tcufg);
    tdp->td_stus = TDSTUS_RUN;//设置进程的状态为运行状态
    krlspinunlock_sti(&tdp->td_lock, &tcufg);
    list_add_tail(&tdp->td_list, &(schdap->sda_thdlst[pity].tdl_lsth));//加入进
    schdap->sda_thdlst[pity].tdl_nr++;
    krlspinunlock_sti(&schdap->sda_lock, &cufg);
    return;
}
```

上面的代码相对简单，我想以你的能力，还能写出比以上更好的代码。好了，到这里，我们进程的等待与唤醒的机制已经实现了。

空转进程

下面我们一起来建立空转进程 ，它也是我们系统下的第一个进程。空转进程是操作系统在没任何进程可以调度运行的时候，就选择调度空转进程来运行，可以说**空转进程是进程调度器最后的选择。**

请注意，这个最后的选择一定要有，现在几乎所有的操作系统，都有一个或者几个空转进程（多 CPU 的情况下，每个 CPU 一个空转进程）。我们的 Cosmos 虽然是简单了些，但也必须要有空转进程，而且这是我们 Cosmos 上的第一个进程。

建立空转进程

我们 Cosmos 的空转进程是个内核进程，按照常理，我们只要调用上节课实现的建立进程的接口，创建一个内核进程就好了。

但是我们的空转进程有点特殊，它是内核进程没错，但它不加入调度系统，而是一个专用的指针指向它的。

下面我们来建立一个空转进程。由于空转进程是个独立的模块，我们建立一个新的 C 语言文件 Cosmos/kernel/krlcpuidle.c，代码如下所示。

复制代码

```
thread_t *new_cpuidle_thread()
{

    thread_t *ret_td = NULL;
    bool_t acs = FALSE;
    adr_t krlstkadr = NULL;
    uint_t cpuid = hal_retn_cpuid();
    schdata_t *schdap = &osschedcls.scls_schda[cpuid];
    krlstkadr = krlnew(DAFT_TDKRLSTKSZ);//分配进程的内核栈
    if (krlstkadr == NULL)
    {
        return NULL;
    }
    //分配thread_t结构体变量
    ret_td = krlnew_thread_dsc();
    if (ret_td == NULL)
    {
        acs = krldelete(krlstkadr, DAFT_TDKRLSTKSZ);
        if (acs == FALSE)
        {
```

```
            return NULL;
        }
        return NULL;
    }
    //设置进程具有系统权限
    ret_td->td_privilege = PRILG_SYS;
    ret_td->td_priority = PRITY_MIN;
    //设置进程的内核栈顶和内核栈开始地址
    ret_td->td_krlstktop = krlstkadr + (adr_t)(DAFT_TDKRLSTKSZ - 1);
    ret_td->td_krlstkstart = krlstkadr;
    //初始化进程的内核栈
    krlthread_kernstack_init(ret_td, (void *)krlcpuidle_main, KMOD_EFLAGS);
    //设置调度系统数据结构的空转进程和当前进程为ret_td
    schdap->sda_cpuidle = ret_td;
    schdap->sda_currtd = ret_td;
    return ret_td;
}
//新建空转进程
void new_cpuidle()
{
    thread_t *thp = new_cpuidle_thread();//建立空转进程
    if (thp == NULL)
    {//失败则主动死机
        hal_sysdie("newcpuilde err");
    }
    kprint("CPUIDLETASK: %x\n", (uint_t)thp);
    return;
}
```

上述代码中，建立空转进程由 new_cpuidle 函数调用 new_cpuidle_thread 函数完成，new_cpuidle_thread 函数的操作和前面建立内核进程差不多，只不过在函数的最后，让调度系统数据结构的空转进程和当前进程的指针，指向了刚刚建立的进程。

但是你要注意，上述代码中调用初始内核栈函数时，将 krlcpuidle_main 函数传了进去，这就是空转进程的主函数，下面我们来写好。

复制代码

```
void krlcpuidle_main()
{
    uint_t i = 0;
    for (;; i++)
    {
        kprint("空转进程运行:%x\n", i);//打印
        krlschedul();//调度进程
    }
    return;
}
```

我给你解释一下，空转进程的主函数本质就是个死循环，在死循环中打印一行信息，然后进行进程调度，这个函数就是永无休止地执行这两个步骤。

空转进程运行

我们已经建立了空转进程，下面就要去运行它了。

由于是第一进程，所以没法用调度器来调度它，我们得手动启动它，才可以运行。其实上节课我们已经写了启动一个新建进程运行的函数，我们现在只要调用它就好了，代码如下所示。

复制代码

```
void krlcpuidle_start()
{
    uint_t cpuid = hal_retn_cpuid();
    schdata_t *schdap = &osschedcls.scls_schda[cpuid];
    //取得空转进程
    thread_t *tdp = schdap->sda_cpuidle;
    //设置空转进程的tss和R0特权级的栈
    tdp->td_context.ctx_nexttss = &x64tss[cpuid];
    tdp->td_context.ctx_nexttss->rsp0 = tdp->td_krlstktop;
    //设置空转进程的状态为运行状态
    tdp->td_stus = TDSTUS_RUN;
    //启动进程运行
    retnfrom_first_sched(tdp);
    return;
}
```

上述代码的逻辑也很容易理解，我为你梳理一下。首先就是取出空转进程，然后设置一下机器上下文结构和运行状态，最后调用 retnfrom_first_sched 函数，恢复进程内核栈中的内容，让进程启动运行。

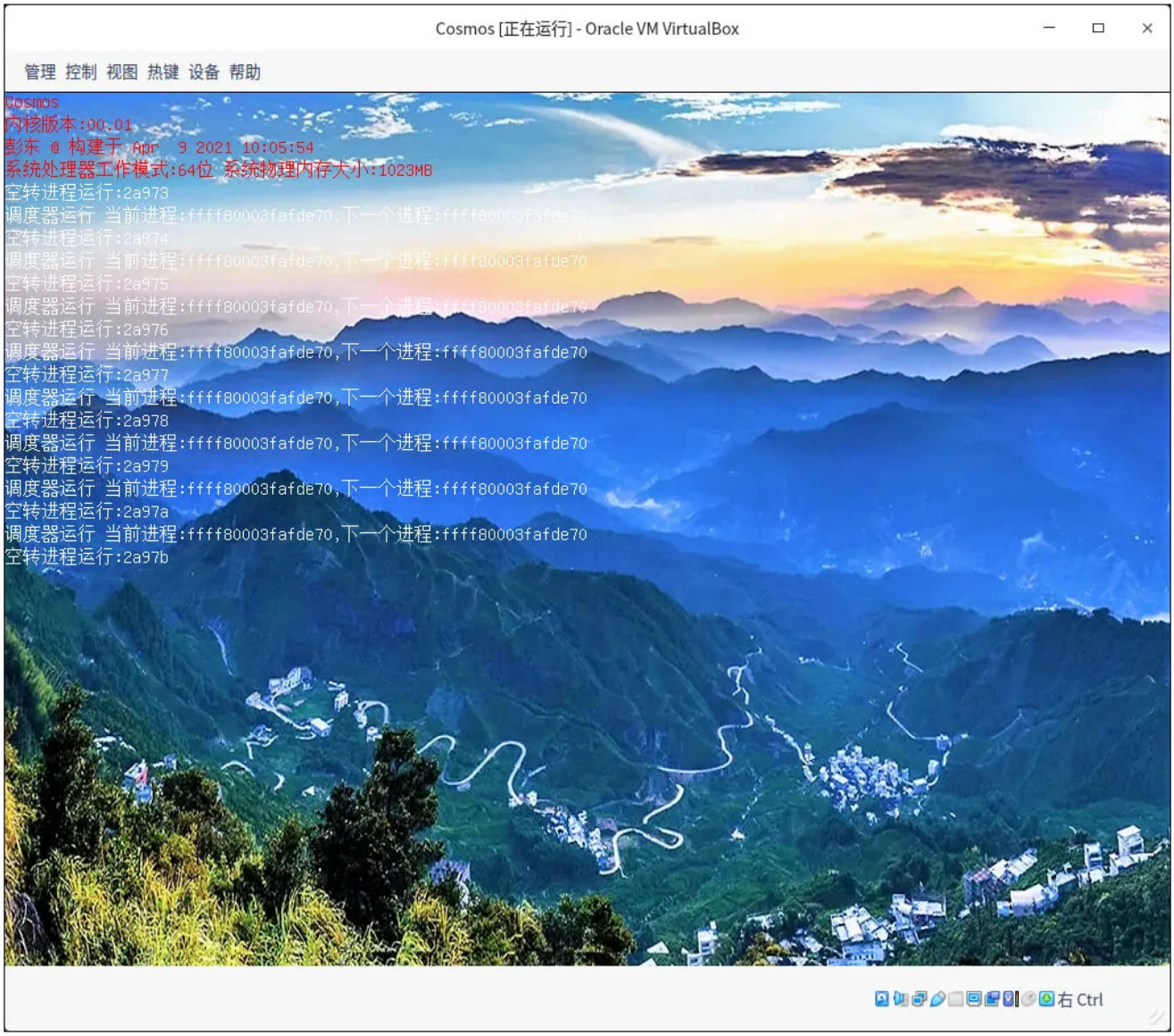
不过这还没完，我们应该把建立空转进程和启动空转进程运行函数封装起来，放在一个初始化空转进程的函数中，并在内核层初始化 init_krl 函数的最后调用，代码如下所示。

复制代码

```
void init_krl()
{
```

```
    init_krlsched();//初始化进程调度器
    init_krlcpuidle();//初始化空转进程
    die(0);//防止init_krl函数返回
    return;
}
//初始化空转进程
void init_krlcpuidle()
{
    new_cpuidle();//建立空转进程
    krlcpuidle_start();//启动空转进程运行
    return;
}
```

好了，所有的代码都已备好，终于到我们检验学习成果的时候了，我切换到这节课程的 cosmos 目录下执行 make vboxtest 命令，就会出现如下图的结果，如下图所示。



空转进程运行

可以看到，现在空转进程和调度器输出的信息在屏幕上交替滚动出现，这说明我们的空转进程和进程调度器都已经正常工作了。

多进程运行

虽然我们的空转进程和调度器已经正常工作了，但你可能心里会有疑问，我们系统中就一个空转进程，那怎么证明我们进程调度器是正常工作的呢？

其实我们在空转进程中调用了调度器函数，然后进程调度器会发现系统中没有进程，又不得不调度空转进程，所以最后结果就是：空转进程调用进程调度器，而调度器又选择了空转进程，导致形成了一个闭环。

但是我们现在想要看看多个进程会是什么情况，就需要建立多个进程。下面我们马上就来实现这个想法，代码如下。

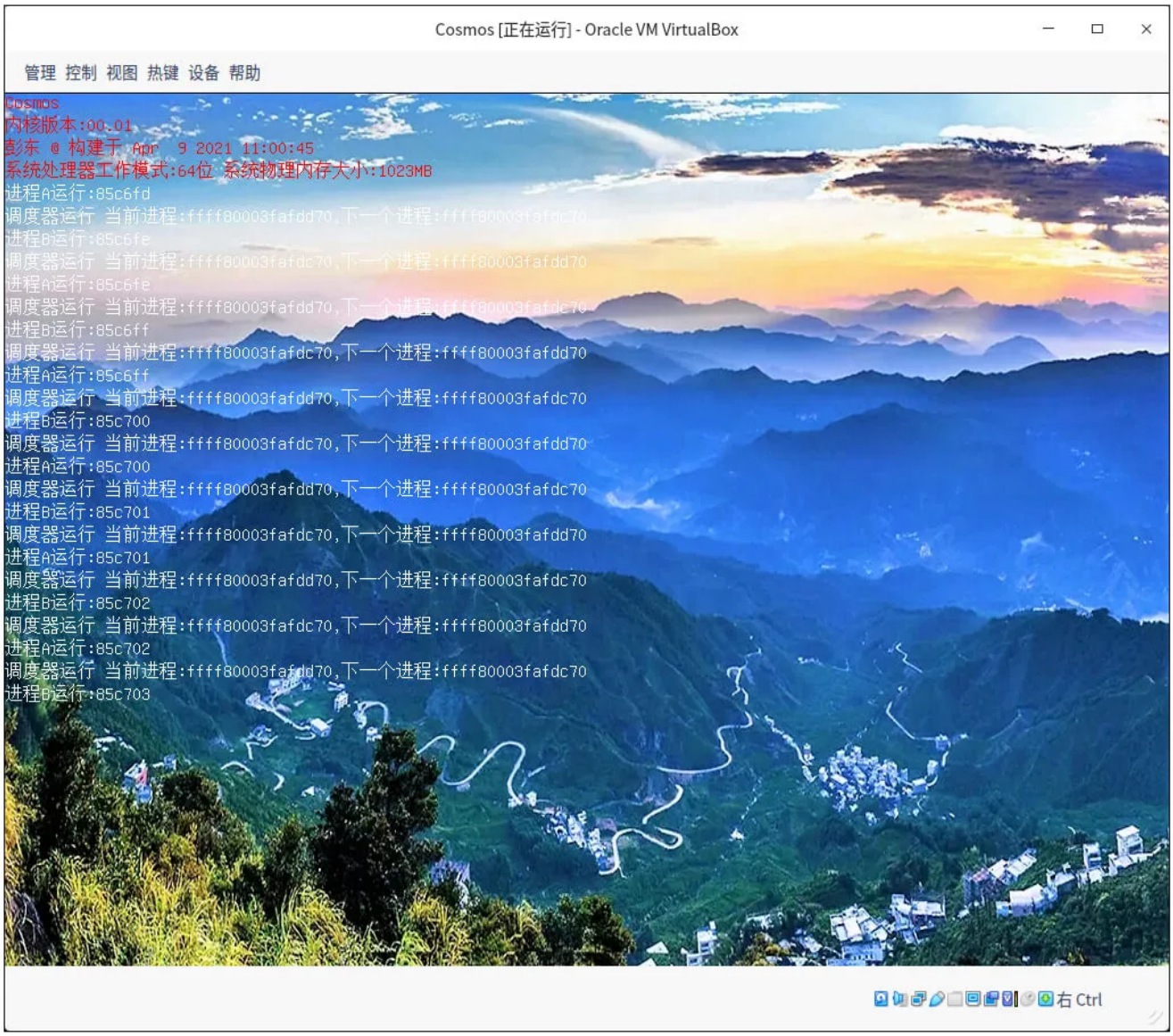
复制代码

```
void thread_a_main()//进程A主函数
{
    uint_t i = 0;
    for (;; i++) {
        kprint("进程A运行:%x\n", i);
        krlschedul();
    }
    return;
}
void thread_b_main()//进程B主函数
{
    uint_t i = 0;
    for (;; i++) {
        kprint("进程B运行:%x\n", i);
        krlschedul();
    }
    return;
}
void init_ab_thread()
{
    krlnew_thread((void*)thread_a_main, KERNTHREAD_FLG,
                PRILG_SYS, PRITY_MIN, DAFT_TDUSRSTKSZ, DAFT_TDKRLSTKSZ);//建立
    krlnew_thread((void*)thread_b_main, KERNTHREAD_FLG,
                PRILG_SYS, PRITY_MIN, DAFT_TDUSRSTKSZ, DAFT_TDKRLSTKSZ);//建立
    return;
}
void init_krlcpuidle()
{
```

```
29      new_cpuidle();//建立空转进程
30      init_ab_thread();//初始化建立A、B进程
31      krlcpuidle_start();//开始运行空转进程
32      return;
33  ⌊
```

上述代码中，我们在 init_ab_thread 函数中建立两个内核进程，分别运行两个函数，这两个函数会打印信息，init_ab_thread 函数由 init_krlcpuidle 函数调用。这样在初始化空转进程的时候，就建立了进程 A 和进程 B。

好了，现在我们在 Linux 终端下进入 cosmos 目录，在目录下输入 make vboxtest 运行一下，结果如下图所示。



两个进程结果截图

上图中，进程 A 和进程 B 在调度器的调度下交替运行，而空转进程不再运行，这表明我们的多进程机制完全正确。

重点回顾

这节课我们接着上一节课，实现了进程的等待与唤醒机制，然后建立了空转进程，最后对进程调度进行了测试。下面我来为你梳理一下要点。

1.**等待和唤醒机制。**为了让进程能进入等待状态随后又能在其它条件满足的情况下被唤醒，我们实现了进程等待和唤醒机制。

2.**空转进程。**是我们 Cosmos 系统下的第一个进程，它只干一件事情就是调用调度器函数调度进程，在系统中没有其它可以运行进程时，调度器又会调度空转进程，形成了一个闭环。

3.**测试。**为了验证我们的进程调度器是否是正常工作的，我们建立了两个进程，让它们运行，结果在屏幕上出现了它们交替输出的信息。这证明了我们的进程调度器是功能正常的。

你也许发现了，我们的进程中都调用了 krlschedul 函数，不调用它就是始终只有一个进程运行了，你在开发应用程序中，需要调用调度器主动让出 CPU 吗？

这是什么原因呢？这是因为我们的 Cosmos 没有定时器驱动，系统的 TICK 机制无法工作，一旦我们系统 TICK 机开始工作，就能控制进程运行了多长时间，然后强制调度进程。系统 TICK 设备我们等到驱动与设备相关的模块，再给你展开讲解。

思考题

请问，我们让进程进入等待状态后，这进程会立马停止运行吗？

欢迎你在留言区和我交流，相信通过积极参与，你将更好地理解这节课的内容。

好，我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 多个活动要安排（上）：多进程如何调度？

下一篇 27 | 瞧一瞧Linux：Linux如何实现进程与进程调度?

精选留言 (7)

写留言

neohope 置顶
2021-07-07

一、数据结构
全局有一个osschedcls变量，其数据结构为schedclass_t，用于管理所有cpu的所有进程。

schedclass_t包括一个 schdata_t数组，每个cpu对应一个。...
展开

作者回复: 不会有问题的 因 SMP下每个CPU会对应一个idle 进程 运行同一份代码

1

pedro 置顶

2021-07-07

并不会，进程进入等待只是进程状态发生了改变，进程还未让出当前CPU的执行权，待调度后，即 krlschedul()，会寻找已经准备好的其它进程，切换CPU上下文，让出CPU，此时该进程才会真正的停止。

所以调度函数至关重要！

展开

作者回复: 66666 对的

2

geek2020

2021-07-15

好奇很多资料说java执行了Thread.sleep()后，会主动让出CPU的使用权，是怎么做到的？

作者回复: sleep 会设置定时器 然后 主动 调用调度器

Feen

2021-07-13

进程进入等待状态后，进程本身处于等待状态，进程被剥夺CPU使用权是通过krlschedul()函数剥夺的，而进程本身并没有调用krlschedul()函数，所以不会立即停止运行，而剥夺CPU使用权是由krlschedul()函数控制的，当保存当前进程寄存器和栈，并且准备好下一个进程的运行函数后，才会让当前进程退出CPU而停止运行了。

展开

作者回复: 对的

青玉白露

2021-07-07

进程不会马上停止运行，调用krlschedul()之后，会调度新的进程。

作者回复: 是的 正确

blentle

2021-07-07

好奇，这个空转进程是我们平时所说的0号进程吗，1号进程的创建是fork它吗

作者回复: 这不一定，我们也不用fork

罗乾林

2021-07-07

进程进入等待状态后，这进程不会立马停止运行，要等到调用krlschedul函数后

作者回复: 对的